RAS-NANO: A Reliability-Aware Synthesis Framework for Reconfigurable Nanofabrics *

Chen He[1] and Margarida F. Jacome
Department of Electrical and Computer Engineering
The University of Texas at Austin
{che,jacome}@ece.utexas.edu
[1]Also with Freescale Semiconductor, Inc.

Abstract

Entering the nanometer era, a major challenge to current design methodologies and tools is to effectively address the high defect densities projected for nanotechnologies. To this end, we proposed a reconfiguration-based defect-avoidance methodology for defect-prone nanofabrics. It judiciously architects the nanofabric, using probabilistic considerations, such that a very large number of alternative implementations can be mapped into it, enabling defects to be circumvented at configuration time in a scalable way. Building on this foundation, in this paper we propose a synthesis framework aimed at implementing this new design paradigm. A key novelty of our approach with respect to traditional high level synthesis is that, rather than carefully optimizing a single ('deterministic') solution, our goal is to simultaneously synthesize a large family of alternative solutions, so as to meet the required probability of successful configuration, or yield, while maximizing the family's average performance. Experimental results generated for a set of representative benchmark kernels, assuming different defect regimes and target yields, empirically show that our proposed algorithms can effectively explore the complex probabilistic design space associated with this new class of high level synthesis problems.

1. Introduction

Emerging nanotechnologies have seen significant advances in recent years [1, 2, 3, 4], and it is predicted that the manufacturing of computing nanosystems is likely to become practical within 10-15 years [5]. Besides the inevitable challenges in terms of complexity and scalability, the ability to handle defective devices will be a critical element of any future architecture, since defect rates are expected to be much higher than current values [3, 5, 6]. Building on the success of the TERAMAC experiment, Heath et. al. identified the possibility of utilizing reconfiguration to achieve defect tolerance in systems targeted at emerging nanotechnologies [6]. Since then, several nanostructures well suited to creating reconfigurable computational fabrics have been successfully demonstrated, see e.g., [1, 2, 4]. Yet, design approaches implementing defect tolerance via reconfiguration have to contend with a major scalability challenge - defect mapping and configuration must be performed on a per chip basis [6, 4, 7, 8]. Recently, we proposed in [9, 10] a probabilistic design paradigm aimed at enabling both such complex tasks to be performed on chip, relying on the processing power of the fabric itself – a critical step towards ensuring scalability. It is based on structuring designs as hierarchies of carefully dimensioned (re)configurable fabric regions, while decomposing and assigning functional *flows* to each region – by restricting the functionality preassigned to a specific nanofabric region, we effectively limit the scope and complexity of the associated defect mapping and configuration tasks, see details in Section 2.

Beyond providing a promising foundation towards addressing the scalability challenge of reconfiguration-based defect-avoidance techniques, the approach in [9, 10] gives also a framework in which to explore critical new tradeoffs among performance, yield, and complexity – as will be seen, the probabilistic nature of these tradeoffs makes this new class of 'reliability-aware' high-level synthesis (HLS) problems quite unique. In particular, rather than carefully optimizing a single ('deterministic') solution, as done in traditional HLS [12], the reliability-aware HLS problem requires the joint synthesis and optimization of a sufficiently large family of alternative solutions, so as to enable actual defects to be circumvented at configuration time - critical towards meeting the target probability of successful configuration, or *yield*. The need to provide 'redundant' (or 'extra') configuration capacity, so as to enable such multiple solutions [9, 10], is inexistent in traditional HLS, and fundamentally impacts system performance, thus leading to a substantial departure on the way the design space should be explored. For example, judiciously increasing the 'size' of the behavioral flows (or 'instructions') to be atomically executed on application-specific functional units, usually critical to achieving high performance in traditional HLS, may actually hurt expected performance in this new context. This is so because 'larger' flows may require substantially more redundant configuration capacity to meet the target yield, thus decreasing the degree of locality of a design, see [9, 10].

Due to those key differences with respect to traditional HLS, this new class of problems requires the definition of a new HLS framework and associated support algorithms, appropriately exploring the design space. This is the topic of this paper. Specifically, we propose a Reliability-Aware Synthesis framework for reconfigurable NANOfabrics (RAS-NANO), aimed at systematically solving the reliability-aware HLS problem defined in [9, 10]. The broad goal of RAS-NANO is to generate designs that achieve the specified target yield with best-expected performance, for a given application kernel.

We start by introducing RAS-NANO's main synthesis flow and then discuss in some detail the algorithms implementing its various phases. Then, relying on extensive experimental data generated for a set of representative benchmark kernels, assuming different defect regimes and target yields, we empirically show that the proposed framework can effectively explore the complex probabilistic design space defined by this new class of HLS problems. When highly dense yet defect-prone nanofabrics are considered, reconfiguration-based defect-avoidance techniques provide a promising direction towards enhancing yield [6, 7, 4, 8], thus the relevance of the form of HLS addressed in this paper.

The paper is organized as follows. In Section 2 we briefly review the architecture of our target nanofabric. An overview of RAS-NANO is given in Section 3, and Sections 4, 5 and 6 discuss in detail the various algorithms implemented in the framework. Experimental results are presented in Section 7. Finally, Section 8 concludes the paper.

2. Background on target nanofabric architecture

The three levels of design abstraction implemented on the programmable nanofabric proposed in [9] are illustrated in Fig.1. The basic configuration unit of the nanofabric, called a *region*, is a grid comprised of eight processing elements (PEs) and eight switching elements (SEs). The PEs perform standard 8-bit arithmetic/logic operations, while the SEs support up to two routing channels among adjacent PEs. Each re-

*This work is supported in part by SRC Grant CRS 1152.001 and NSF Grant CCR 0310119.

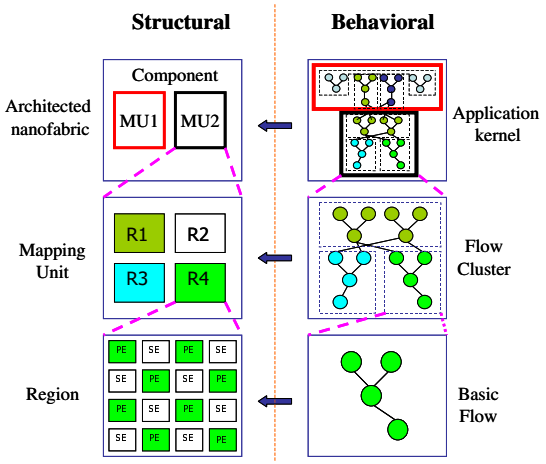

Figure 1. Hierarchy of design abstractions.

gion of the nanofabric can be configured to execute a small behavioral segment, or 'complex instruction', called a *basic flow* – see bottom part of Fig.1. In order to enhance the probability of successful configuration when defects are present in the target region, basic flows must contain less than eight operations/nodes – for example, the basic flow shown on the bottom frame of Fig.1 has only four operations. In order to further increase such probability in a scalable way, the fabric architecture provides an additional hierarchical level, comprised of *mapping units* (MUs) – see middle part of Fig.1. Namely, m basic flows, i.e., a *flow cluster*, can be instantiated in an MU containing n regions, where m is less than or equal to n – the middle frame of Fig.1, for example, shows a flow cluster with three basic flows being mapped to an MU with four regions.[1] Finally, MUs are grouped together to form a *component* of the nanofabric, implementing an application kernel, e.g., the auto-regression filter shown on the top of Fig.1 is comprised of two MUs – for more details, see [9, 10].

3. Overview of reliability-aware synthesis flow

Given an application kernel, the goal is to generate a design with 'best-expected' performance meeting the specified probability of successful configuration, or yield. Fig. 2 shows the overall synthesis flow implemented in RAS-NANO – for simplicity, the target yield for intra- and inter-MU communication resources is specified separately from the yield for the component's transformational resources, comprising the actual regions instantiated within its MUs.

As shown in Fig. 2, the main steps of RAS-NANO's synthesis flow are:

(1) *RAS-BehavBounds*: decide on the maximum number of operations (or nodes) allowed on each basic flow ('instruction size'), and on the maximum number of basic flows possible to map into a single mapping unit (MU) – discussed in Section 4.

(2) *RAS-Allocation*: decide on the number of MUs instantiated in the component, and on the number of regions within each such MU – discussed in Section 4.

(3) *RAS-InstructionSelection*: generate a flow cover for the kernel's dataflow graph (DFG) using basic flows with no more than the allowed number of nodes – discussed in Section 5.

(4) *RAS-Binding*: cluster and assign basic flows to MUs, subject to convexity constraints (unlike conventional HLS, this is a many-to-many binding) – discussed in Section 5.

(5) *RAS-Routing*: specify the communication structure, i.e., the number of intra- and inter-MU tracks instantiated in the nanofabric – discussed in Section 6.

The decisions made in steps RAS-BehavBounds and RAS-Allocation are based on rough, preliminary yield estimates – Section 4 discusses

[1] As discussed in [9], the *mapping unit* abstraction creates a second level of redundancy while retaining the original simplicity of the region based defect mapping and configuration.

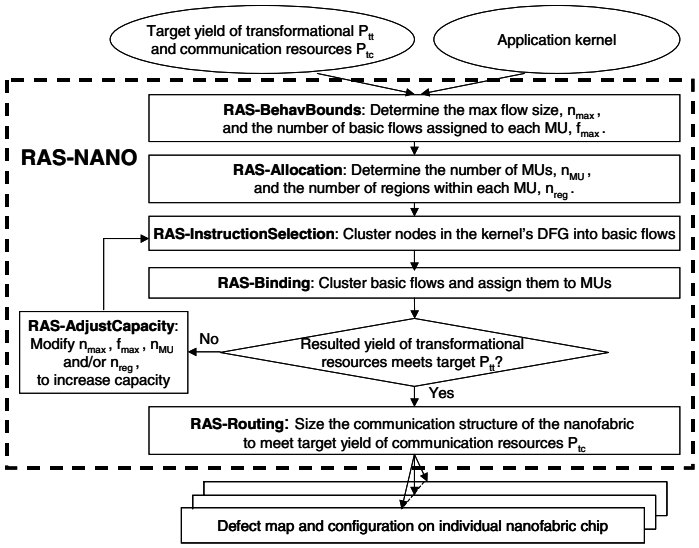

Figure 2. RAS-NANO: overall high-level synthesis flow.

how such estimates are generated. Once all design details are fully defined, one still needs to check if the actual yield meets the target value. As shown in Fig. 2, if it does not, the level of redundant configuration capacity provided on the design needs to be increased (in *RAS-AdjustCapacity*) and then a new design cycle is initiated.

Final Configuration Step. Once the design is finished and a corresponding batch of programmable nanofabric chips are fabricated, those chips still need to be configured. Namely, the defects on each chip need to be first individually mapped (using the TMR-tile based group testing method described in [11]) and then, based on such results, an exact placement of each basic flow onto a region of the appropriated MU is determined, such that the identified defective resources are avoided – this topic was extensively addressed in [9, 10] and is beyond the scope of this paper.

4. RAS-BehavBounds and RAS-Allocation

The first two steps of RAS-NANO's synthesis flow define the level of 'redundant' configuration capacity should be provided in the nanofabric, so as to achieve the target yield. Specifically, they define the 'size' of the behavioral abstractions (basic flows and clusters of flows) and the 'size' of the corresponding structural abstractions (number of MUs and number of regions per MU), as discussed below.

4.1. Algorithm to determine configuration capacity

In order to enhance a component's yield, one should increase its level of 'redundant' configuration capacity, by: 1) instantiating more regions in its MUs; 2) using a kernel cover with smaller basic flows; and/or 3) instantiating more MUs and assigning fewer basic flows to each MU.[9, 10] The previous alternatives are listed in increasing order of their impact on yield and expected component latency – see experimental results in [9, 10]. Accordingly, since our goal is to meet the yield constraint with best expected performance, we have developed a greedy algorithm that increases configuration capacity, in that order, until the target yield is met.

The pseudo-code of our algorithm is shown in Fig.3. Its inputs are: n_G – number of nodes in kernel's DFG G; and P_{tt} – target yield for the component's transformational resources. The algorithm's outputs are: n_{max} – maximum number of operations allowed on any basic flow;[2] f_{max} – maximum number of basic flows that can be mapped into one MU; n_{MU} – number of MUs to be instantiated in the component; and n_{reg} – number of regions in each MU.[3] As shown in Fig.3, the algorithm starts by using the least possible number of MUs, the largest

[2] As mentioned before, $n_{max} \leq MaxNodesPerFlow = 7$.

[3] To control routing complexity, $n_{reg} \leq MaxRegionsPerMU = 9$, see [9].

Algorithm RAS-BehavBounds&Allocation:

1. $n_{MU} = max(1,(n_G/MaxRegionsPerMU)/MaxNodesPerFlow))$;
2. while($n_{MU} \leq n_G$) {
3. $n_{max} = MaxNodesPerFlow$;
4. while($n_{max} \geq 1$) {
5. $f_{max} = max(1,(n_G/n_{MU})/n_{max})$; //estimate # of flows per MU
6. $n_{reg} = f_{max}$; //# of regions starts with # of flows per MU
7. while($n_{reg} \leq MaxRegionsPerMU$) {
8. estimate component yield P;
9. if $P < P_{tt}$
10. $n_{reg} = n_{reg} + 1$; //increase configuration capacity 1)
11. else
12. return ($n_{max}, f_{max}, n_{MU}, n_{reg}$); //found the solution
13. }
14. $n_{max} = n_{max} - 1$; //increase configuration capacity 2)
15. }
16. $n_{MU} = n_{MU} + 1$; //increase configuration capacity 3)
17. }
18. return error; //no feasible solution found

Figure 3. Algorithm for RAS-BehavBounds and RAS-Allocation.

possible basic flows, and the minimum number of regions within an MU, and then gradually increases configuration capacity, following the order specified above, until the target yield is met, or unfeasibility is detected. As one would expect, the same order is also used for incremental increases in configuration capacity across design cycles or iterations, within step RAS-AdjustCapacity.

4.2. Yield estimation

Preliminary Yield Estimation. The algorithm in Fig.3 requires estimating the component yield P (see line 8) – we solve this hierarchically, by estimating the yield at each level of the design hierarchy. First, similarly to [9, 10], we estimate basic flow yield at the region level using Monte Carlo simulation, for a specific defect regime (P_e, P_a, P_c), where P_e, P_a, P_c denote the probabilities of failure for PEs; PEs operating as arbiters; and connections, respectively. Specifically, we generate a large number of defect realizations on a region (so as to achieve an adequate confidence level), and then use the TMR-based group testing method described in [9, 10] to obtain a (partial) defect map for each such region instance. We then use a simple table-look-up algorithm to find if a feasible configuration for the basic flow mapped into that region instance exists – the probability of successful configuration, or yield, of the basic flow is given by the actual fraction of region instances for which a feasible configuration has been found.

We run such Monte Carlo simulations for essentially all possible basic flows of various sizes, considering different defect regimes. Fig. 4 shows a sample of our results – namely, minimum and maximum basic flow yields, for basic flows containing one to seven nodes, assuming defect regime $(P_e, P_a, P_c) = (10,5,1)\%$. As one would expect, basic flow yield decreases as the number of nodes in the basic flow increases, yet there is some variation for basic flows of identical size, caused by their distinct connectivity requirements. When initially estimating the yield for a basic flow of a given size, one may select more or less conservative values, depending on how aggressive one may wish to optimize performance, knowing that by choosing less conservative values one may need to iterate over several design cycles.

Yield at the next level of hierarchy, i.e., MU level, is roughly estimated assuming that all basic flows have the same size.[4] For this special case of f_{max} *identical* basic flows being mapped into n_{reg} regions, the MU level yield is directly given by:

$$P_{MU} = \sum_{i=f_{max}}^{n_{reg}} \binom{n_{reg}}{i} P_r^i (1-P_r)^{n_{reg}-i} \quad (1)$$

[4] As before, we assume an independent distribution of defects across regions.

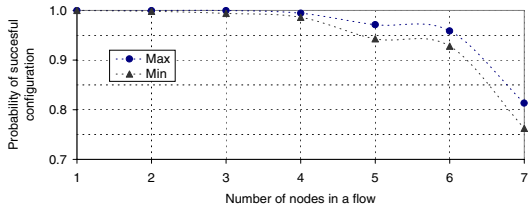

Figure 4. Probability of successful configuration of a basic flow on a region versus flow size when $(P_e, P_a, P_c) = (10,5,1)\%$.

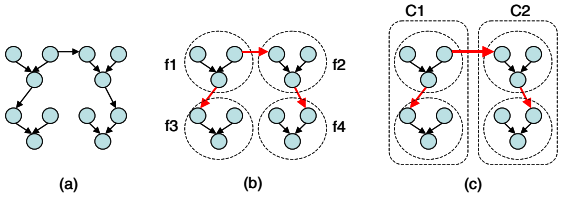

Figure 5. Dataflow model: (a) original DFG G; (b) node-clustered DFG G_f; (c) flow-clustered DFG G_{fc}.

where P_r is the estimated yield for the particular basic flow being considered, obtained as discussed before.

Finally, the yield estimate at the component level P, is given by

$$P = \prod_{i=1}^{n_{MU}} P_{MU_i} \quad (2)$$

where P_{MU_i} is the yield of the component's ith MU.

Final yield estimation. The procedure used to estimate the yield of a detailed component design is very similar to that used during the preliminary phase, except that for the former we know the exact flow cover and resource allocation and assignment decisions implemented in the design, and can thus be more accurate. Also, at the MU level, we need now to consider the more generic case of mapping different basic flows to an MU – we have developed a set of dominance rules to enable a more efficient estimation process for those cases, yet space limitations preclude us from presenting those rules here.

5. RAS-InstructionSelection and RAS-Binding

In the two subsequent steps of RAS-NANO's synthesis flow – also called 'clustering' phase – a flow cover for the kernel's data flow graph (DFG) is generated (*RAS-InstructionSelection*), and then an assignment of the resulting basic flows to MUs is performed (*RAS-Binding*), satisfying the yield related behavioral bounds n_{max} and f_{max}, and attempting to minimize expected (or average) latency. Furthermore, as discussed in [9], the sets of basic flows (or 'flow clusters') assigned to the various MUs must also satisfy convexity constraints, that is, there cannot be 'circular' (input/output) data dependencies among them.

Fig. 5 illustrates the results of the two clustering steps – the output of the RAS-InstructionSelection step is a *node-clustered DFG*, denoted G_f (see basic flows f_1, f_2, f_3 and f_4 in Fig. 5(b)), and the output of the RAS-Binding step is a *flow-clustered DFG*, denoted G_{fc} (see flow clusters C_1 and C_2 in Fig. 5(c)), where each flow cluster is assigned to an MU. Note that G_{fc} has three types of edges: intra-flow edges, inter-flow edges, and inter-flow-cluster edges. As discussed in [9], the intra-flow edges do not cause extra delay, while the inter-flow and inter-flow-cluster edges do incur data transfer delays, corresponding to moving data between regions belonging to the same or to different MUs. Note finally that the delay incurred by such data transfers may vary among different component instances, since the mapping of basic flows to regions is not fixed, and depends upon the actual defect distributions on each chip. Accordingly, our clustering algorithm uses average values for such delays, derived using a combination of analysis and simulation, for different structural scenarios.

Algorithm RAS-Init:

```
1.    for each node v in G following the ranking order {
2.        for each candidate basic flow f {
3.            if cluster(v) = f satisfies size and convexity constraints
4.                calculate trcost(v, f);
5.            else
6.                mark cluster(v) = f inadmissible;
7.        }
8.        select cluster(v) = f minimizing trcost(v, f);
9.        if no admissible clustering of v can be found {
10.           find the first clustered node v_b that satisfies the conditions:
11.               a) cluster(v_b) = f such that cluster(v) = f is non-convex;
12.               b) ∃ a v_b's successor s such that cluster(s) ≠ cluster(v_b);
13.               c) v_b has not been backtracked before;
14.           re-cluster v_b such that cluster(v_b) = cluster(s);
15.           go back to line 1 and loop restarts from the node after v_b;
16.           if such backtrack node v_b cannot be found
17.               create a new basic flow for v to be clustered;
18.   } }
```

Figure 6. Algorithm for initial reliability-aware clustering.

5.1. Algorithm for clustering phase: RAS-TPC

RAS-NANO's clustering phase bears considerable resemblance to clustering problems defined in the context of traditional HLS and compilers, see e.g., [15, 16, 13, 14]. In fact, we were able to successfully adapt TPC (Two-Phase Clustering), a well-known state-of-the-art algorithm proposed by Lapinskii *et. al.*[15], to address our clustering phase. As discussed in the sequel, our 'reliability-aware' version of the algorithm, denoted RAS-TPC, is used in both the node clustering and the flow clustering phases of RAS-NANO.[5].

5.1.1. Node clustering step: RAS-InstructionSelection

Similarly to TPC, RAS-TPC starts by performing a fast greedy clustering, and then iteratively improves on that initial solution – both phases of the algorithm are briefly discussed below.

1) Initial Clustering Phase. The greedy algorithm that generates the initial clustering is shown in Fig.6 – lines in bold represent our additions/enhancements to the original TPC. The order in which nodes are considered for clustering (line 1) is determined by a ranking function identical to that proposed in [15], which considers the as-late-as-possible (ALAP) value of the candidate operation/node, and its mobility and number of successors (see [15] for more details). Then, for the selected node v, we evaluate each possible alternative clustering to a basic flow f ($cluster(v) = f$), using cost function $trcost(v, f)$ (line 4), which similarly to [15], is defined as follows:

$$trcost(v, f) = trcost_{dd}(v, f) + trcost_{cc}(v, f) \quad (3)$$

where the 'direct data dependency' component $trcost_{dd}(v, f)$ favors solutions that place consumer and producer operations into the same basic flow, and the 'common consumer' component $trcost_{cc}(v, f)$ favors solutions that place multiple producers to a common consumer into the same basic flow – for more details see [15].

However, differently from the original TPC, we perform a convexity constraint check during the node clustering step (line 3). Note first that non-convex data dependencies among basic flows assigned to the same MU are allowed, in order to preserve fine-grain parallelism [9]. That is, convexity constraints need to be enforced only across MUs. However, the joint consideration of n_{max} and f_{max} does limit the maximum number of nodes that can be mapped into a single MU – to $n_{max} \cdot f_{max}$. Therefore, although allowing non-convex data dependencies during this phase, we need to make sure that the total number of nodes in basic flows exhibiting 'circular' data dependencies does not exceed that limit, so as to enable all such basic flows to be later mapped to a single MU – otherwise, convexity constraints at the MU level would be violated.

[5]We have also developed a reliability-aware version of HP's Partial Component Clustering (PCC) algorithm [16], denoted RAS-PCC, yet our version of RAS-TPC consistently outperformed RAS-PCC, and thus we only present results for the former.

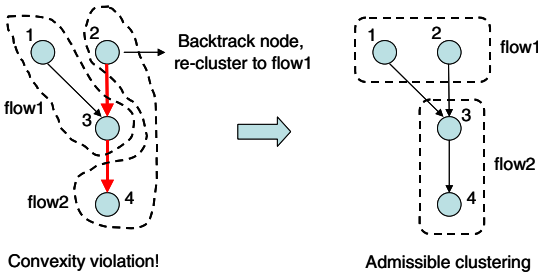

Figure 7. Backtracking during initial clustering.

We use the depth-first search based algorithm [17] to check for convexity constraints' violations. If one such violation is detected, it must be eliminated. Meeting convexity constraints with a greedy clustering algorithm is somewhat challenging, since convexity is a global constraint, while our algorithm makes clustering decisions greedily/locally. In [13] and [14], convexity constraints are considered during instruction selection, yet both algorithms have worst-case exponential computing complexity, in contrast to our low cost heuristic. We have thus developed a simple backtracking strategy to tackle the issue (see line 9 - 15), which has performed quite effective in practice. Fig.7 is used to illustrate the heuristic. Consider a DFG G containing 4 nodes, and assume $n_{max} = 2$, $f_{max} = 1$. After clustering node 1 to basic flow 1, node 2 to basic flow 2, and node 3 to basic flow 1, when the algorithm tries to cluster node 4, no admissible clustering can be found, since clustering it to basic flow 1 violates the flow size constraint, and clustering it to basic flow 2 violates convexity constraints at the MU level. In order to handle the problem, our algorithm starts by backtracking to a previously clustered node, selected as specified in lines 11 to 13. Specifically, following backwards the ranking order for the previously clustered nodes, the algorithm backtracks to the first node v_b that meets the following three conditions: 1)v_b was clustered to a basic flow that the current node cannot be clustered to, without violating convexity constraints ; 2) at least one of v_b's successors, denoted s, was clustered to a basic flow different from its own; 3) v_b was not backtracked to before. Once the backtracking node v_b is selected, the algorithm re-clusters it to the basic flow of its successor s, and then restarts the regular greedy algorithm from the next node in the ranking order. For the example in Fig.7, node 2 would be the first one to satisfy the three conditions, and hence would be selected to be the backtracking node. Node 2 would then be re-clustered to basic flow 1, and the normal clustering process would resume with node 3, eventually generating the solution shown in Fig.7.

Of course, there is always the possibility that the backtracking heuristic cannot generate a feasible solution – either because a convex clustering (at the MU level) does not actually exist or because the heuristic has failed. If this happens, as shown in line 16 and 17, we cluster the problematic node to a new basic flow, in order to avoid size and convexity constraint violations. Of course, this might adversely impact performance, since the use of 'smaller' basic flows results in more inter-flow data transfer delays. Still, our heuristic has so far proven to be quite effective in identifying those clustering decisions that may have caused a constraint violation – specifically, for all of our experiments, it has failed to backtrack successfully only once (for one of the DCT-DIT experiments, see Section 7), resulting in one more basic flow than expected for that case, with insignificant impact on performance.

2) Iterative Improvement Phase. Although our initial clustering algorithm performs pretty well (see Section 7), improvement is in general still possible. To take advantage of these opportunities, similarly to [15], we have developed a relatively low-cost iterative improvement algorithm – see Fig.8, where lines in bold represent our enhancements to the original TPC algorithm.

The iterative improvement algorithm is based on *boundary permutations* [15]. The boundary nodes (line 5) comprise those nodes

Algorithm RAS-IterativeImprovement:

```
1.    progress = 0; iteration = 0;
2.    compute the initial clustering cost;
3.    do {
4.        iteration++;
5.        for each boundary node v in G {
6.            for each node p ∈ adj(v) and cluster(p) ≠ cluster(v) {
7.                temporarily move v to cluster(p);
8.                makes a chain of temporary moves of boundary nodes
                  until target flow contains no more than n_max nodes;
9.            }
10.           if the temporary clustering satisfies convexity constraint {
11.               compute the new clustering cost;
12.               if the clustering cost improves {
13.                   commit the chain of moves and update the clustering;
14.                   progress = 1; }
15.       } }
16.   } while (progress == 1)and(iteration <= iteration_max);
```

Figure 8. Algorithm of reliability-aware iterative improvement for clustering.

that have either predecessors or successors clustered to different basic flows – those nodes will be moved around different basic flows, providing opportunities for eliminating or collapsing associated inter-flow data transfers. Differently from the original TPC algorithm, though, our boundary permutations need to satisfy constraint n_{max}, as well as convexity constraints. Namely, after moving a boundary node to a different basic flow, the latter may contain more than n_{max} nodes, and hence we need to make sure that it will also export a boundary node to another basic flow – such *chain of moves* should continue until the last basic flow to receive a boundary node still contains no more than n_{max} nodes – see lines 7 and 8 in Fig.8. Note also that, at any step of the chain, if there are multiple options, the one minimizing cost function (4) (discussed below) is selected.

After completing one such chain of moves, we obtain a new temporary clustering solution and evaluate it using a suitable cost function (line 11) – if the resulting cost improves, we accept the chain of moves and update the current clustering solution. The algorithm terminates when all possible chains of moves fail to improve cost or an upper bound on the number of iterations is reached. The following cost function is used in RAS-TPC:

$$C_{cost} = (L_{G_f}, N_m, M_b) \tag{4}$$

where L_{G_f} is the ASAP schedule latency of the node-clustered DFG G_f, N_m is the number of inter-flow edges, and M_b is the solution's mobility balance. To compare two clustering solutions, the three components of the cost function are compared in lexicographical order. L_{G_f} and N_m aim at minimizing average latency. M_b aims at discouraging the clustering of nodes with large mobility differences into the same basic flow, since this will likely decrease the exposed instruction level parallelism, and thus potentially harm performance as well[6] – it is defined as the sum of mobility differences over all flows, i.e.,

$$M_b = \sum_{\forall f \in G_f} \mu_{f,max} - \mu_{f,min} \tag{5}$$

where $\mu_{f,max}$ and $\mu_{f,min}$ denote the maximum and minimum mobility associated to the nodes in basic flow f, respectively.

5.1.2. Flow clustering step: RAS-Binding

After clustering the original DFG G into basic flows, as discussed above, we contract each resulting basic flow to a node v_c, and construct a corresponding *contracted graph* G_c.[7] When there is a 'circular' data dependency among basic flows (this is possible since non-convexity is allowed at the node clustering phase), we further contract all the flows contained in the 'circular' dependency path into a single node (such that all such basic flows will necessarily be mapped to a single MU), and the contracted graph G_c becomes acyclic. (Recall that, as discussed in Section 5.1.1, the actual number of basic flows contained in each such contracted node will necessarily satisfy f_{max}.) After generating the acyclic contracted graph G_c, we simply re-apply the RAS-TPC algorithm (used in the previous step) to G_c, so as to derive the set of basic flows (or flow clusters) to be assigned to each MU, where cluster size is now limited by f_{max} (rather than n_{max}).

6. RAS-Routing

After decisions on MU allocation and binding are made, we still need to determine how many routing tracks are needed to support intra- and inter-MU data transfers, assuming a target nanofabric with uniform routing channels [9, 10], i.e., with the same number of tracks on all channels, and considering a given probability of a track being defective (P_{et}). Note that, differently from previous approaches, the exact placement of basic flows into the internal regions of a component's MUs is not known – our goal is to actually determine the number of tracks required to support all potential alternative solutions. In order to do so, we consider a number of distinct basic flow placements, aimed at exposing different forms of 'congestion', namely: 1) 'compact' placement – the basic flows are mapped to regions all located at one corner of the corresponding MU; 2) 'spread' placement – the basic flows are mapped to regions as far from each other as possible on the appropriated MU; and 3) random placements – the basic flows are randomly mapped to regions of the appropriated MU. For each such placement, we start by assuming that all routing tracks are defect-free, thus reducing our problem to the so called 'symmetrical FPGA array routing'.[18] We then execute the well-known Pathfinder congestion negotiation routing algorithm, used in VPR [18], to obtain the number of tracks required by that particular solution. Once the process is repeated for all alternative placements, we select the highest number of defect-free routing tracks required by any such solution. Then, given P_{et} and the target yield for communication resources P_{tc}, we use a binomial distribution (assuming that all tracks are probabilistically independent from a standpoint of faults) to estimate the number of tracks required in the presence of defects.

7. Experimental validation

Table 1 shows the characteristics of the benchmarks used in our experiments. They include an auto-regression filter (AR), an avenhous filter (AF), a finite impulse response filter (FIR) and its unrolled version (FIRu), an elliptic wave filter (EWF), a version of the fast fourier transform (FFT) used in [15], and various discrete-cosine transform (DCT) algorithms [15].

Table 1. Characteristics of benchmark kernels.

Kernels	#nodes	#nodes in critical path	#connected components
AR	28	8	1
AF	18	7	1
FIR	16	9	1
FIRu	32	11	1
EWF	34	14	1
FFT	38	4	3
DCT-LEE	49	9	2
DCT-DIF	41	7	2
DCT-DIT	48	7	1

Table 2 shows a sample of the results generated by RAS-NANO for each kernel, for various values of P_{tt} and P_{tc}, and assuming defect regime $(P_e, P_a, P_c) = (10, 5, 1)\%$ (Results for other defect regimes exhibit similar trends and are omitted due to space limitations.) For each design, n_{max}, f_{max}, n_{MU} and n_{reg} are shown in columns 3 to 6, respectively.[8] The design's average latency is given in column 9 – this

[6] Since an MU cannot start execution until all of its input data is ready[9], nodes with high mobility will have to wait until the data for the low mobility nodes arrives, if such data is produced by a different MU.

[7] There is an edge between two nodes in G_c if there is a data dependency between any pair of nodes in the corresponding two basic flows.

[8] Recall that those denote the maximum basic flow size, maximum number of flows

value is computed assuming a two cycle operation delay[9, 10], and average inter-flow and inter-flow-cluster data transfer delays of two and four cycles, respectively. The number of required routing tracks (to support inter- and intra-MU data transfers) is given in columns 7 and 8, where the former (*ReqTracks*) is the defect free value (worst case among 8 different basic flow placement scenarios, as described in Section 6), and the latter (*ActualTracks*) is the total number of tracks required to meet the target yield P_{tc}, assuming the probability of a track being defective $P_{et} = 1\%$.

Table 2. Results for benchmarks when $(P_e, P_a, P_c) = (10, 5, 1)\%$.

Kernel	P_{tt} (=P_{tc})	n_{max}	f_{max}	n_{MU}	n_{reg}	ReqTracks	ActualTracks	AvgLatency
AR	$1-10^{-5}$	6	3	2	8	5	8	22
	$1-10^{-10}$	4	3	3	9	4	9	26
	$1-10^{-15}$	3	2	5	9	3	11	30
AF	$1-10^{-5}$	6	3	1	8	6	9	16
	$1-10^{-10}$	4	3	2	9	5	11	22
	$1-10^{-15}$	3	2	3	9	3	11	26
FIR	$1-10^{-5}$	6	3	1	8	4	7	22
	$1-10^{-10}$	4	2	2	7	3	8	26
	$1-10^{-15}$	3	2	3	9	3	11	30
FIRu	$1-10^{-5}$	6	3	2	8	6	9	30
	$1-10^{-10}$	4	3	3	9	4	9	34
	$1-10^{-15}$	3	2	6	9	4	12	44
EWF	$1-10^{-5}$	6	3	2	8	6	9	34
	$1-10^{-10}$	4	3	3	9	6	12	38
	$1-10^{-15}$	3	2	6	9	4	12	44
FFT	$1-10^{-5}$	5	4	2	9	7	10	16
	$1-10^{-10}$	4	3	4	9	5	11	22
	$1-10^{-15}$	3	2	7	9	4	12	30
DCT-LEE	$1-10^{-5}$	6	3	3	9	6	9	32
	$1-10^{-10}$	4	3	5	9	5	11	40
	$1-10^{-15}$	3	2	9	9	3	11	52
DCT-DIF	$1-10^{-5}$	4	6	2	9	7	10	28
	$1-10^{-10}$	4	3	4	9	5	11	34
	$1-10^{-15}$	3	2	7	9	4	12	48
DCT-DIT	$1-10^{-5}$	4	6	2	9	9	12	24
	$1-10^{-10}$	4	3	4	9	6	12	38
	$1-10^{-15}$	3	2	8	9	4	12	50

The results presented in Table 2 empirically show that the RAS-NANO framework can effectively explore the complex probabilistic design space defined by the reliability-aware HLS problem addressed in this paper. Namely, as it can be seen, average component latency increases with increases in target yield – this is exactly what one would expect, since increases in redundant configuration capacity are needed to achieve the higher yield, but they have a deleterious effect on locality. Note also that the number of *defect free* tracks decreases with increases in yield, since increasing redundant configuration capacity leads to less congested designs. However, when we simultaneously (and more realistically) consider the presence of defective routing tracks, again the number of tracks tends to increase as the target yield increases.

RAS-NANO's execution time is essentially determined by its clustering phase. Thus, to assess the effectiveness of the somewhat costly iterative improvement step of RAS-TPC, we considered the complete algorithm as well as the version without iterative improvement, i.e., RAS-INIT. For each kernel, we report the execution time of both algorithms, as well as the average latency of their corresponding design solutions. Fig.9 shows samples of our results, generated for target yield P_{tt} of $1-10^{-15}$. The execution time in milliseconds was obtained on a SparcV9 750MHz processor. In average over all our experiments, RAS-TPC achieves 17% improvement in average latency with respect to RAS-INIT, with roughly 4 times increased execution time. Accordingly, in the current version of RAS-NANO, designers can select to enable iterative improvement or not, based on their specific applications, optimization goal, and sensitivity to execution time.

mapped to one MU, the number of MUs, and the number of regions per MU, for the particular design.

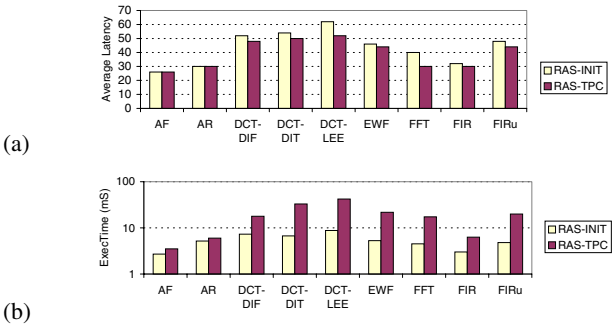

Figure 9. Results for benchmarks with target yield $P_{tt} = 1-10^{-15}$ when $(P_e, P_a, P_c) = (10, 5, 1)\%$: (a) Average latency; (b) Algorithm execution time in milliseconds.

8. Conclusions

In this paper, we have proposed a reliability-aware high-level synthesis (HLS) framework, RAS-NANO, aimed at synthesizing a sufficiently large family of alternative solutions, to be mapped on defect-prone but reconfigurable nanofabrics, so as to meet the specified yield with best expected performance. Experimental results generated for a set of representative benchmark kernels, assuming different defect regimes and target yields, empirically show that this synthesis framework can effectively explore the complex design space induced by the new class of reliability-aware HLS problems.

References

[1] C. P. Collier et. al., "Electronically configurable molecular-based logic gates," *Science*, vol. 285, pp. 391–94, July 1999.

[2] T. Rueckes et. al., "Carbon nanotube based non-volatile random access memory for molecular computing," *Science*, vol. 289, pp. 94–97, 2000.

[3] G. Bourianoff, "The future of nanocomputing," *IEEE Computer*, pp. 44–49, Aug. 2003.

[4] A. DeHon, "Array-based architecture for FET-based nanoscale electronics," *IEEE Trans. Nanotechnology*, vol. 2, no. 1, pp. 23–32, 2003.

[5] SEMATECH, "International Technology Roadmap for Semiconductors," http://www.itrs.net/Common/2004Update/2004Update.htm.

[6] J. R. Heath et. al., "A defect-tolerant computer architecture: Opportunities for nanotechnology," *Science*, vol. 280, pp. 1716–21, June 1998.

[7] M. Mishra and S. C. Goldstein, "Defect tolerance at the end of the roadmap," in *Proc. ITC*, 2003.

[8] J. Han and P. Jonker, "A defect- and fault-tolerant architecture for nanocomputers," *Nanotechnology*, vol. 14, pp. 224–230, 2004.

[9] M. F. Jacome, C. He, G. de Veciana, and S. Bijanski, "Defect tolerant probabilistic design paradigm for nanotechnologies," in *Proc. DAC*, 2004.

[10] C. He, M. F. Jacome, and G. de Veciana, "A reconfiguration-based defect-tolerant design paradigm for nanotechnologies," *IEEE Design & Test of Computers*, vol. 22, no. 4, pp. 316–326, July-August 2005.

[11] C. He, M. F. Jacome, and G. de Veciana, "Scalable defect mapping and configuration of memory-based nanofabrics," in *Proc. HLDVT*, 2005.

[12] W. Geurts et. al., *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*, Kluwer Academic Publishers, 1997.

[13] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. DAC*, 2003.

[14] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. CASES*, 2004.

[15] V. S. Lapinskii, M. F. Jacome, and G. de Veciana, "Cluster assignment for high-performance embedded VLIW processors," *ACM Trans. Design Auto. of Elec. Sys.*, vol. 7, pp. 430–454, 2002.

[16] G. Desoli, "Instruction assignment for clustered VLIW DSP compilers," Tech. Rep., Hewlett-Packard Company, 1997, Tech. Report HPL-98-13.

[17] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal of Computing*, vol. 1(2), pp. 146–160, 1972.

[18] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Workshop on FPLA*, 1997.